Joan McCarthy
12744729

COMP20230 - Data Structures & Algorithms
Project – Zebra Puzzle

1.0 Project Structure

1.1 UML Diagrams

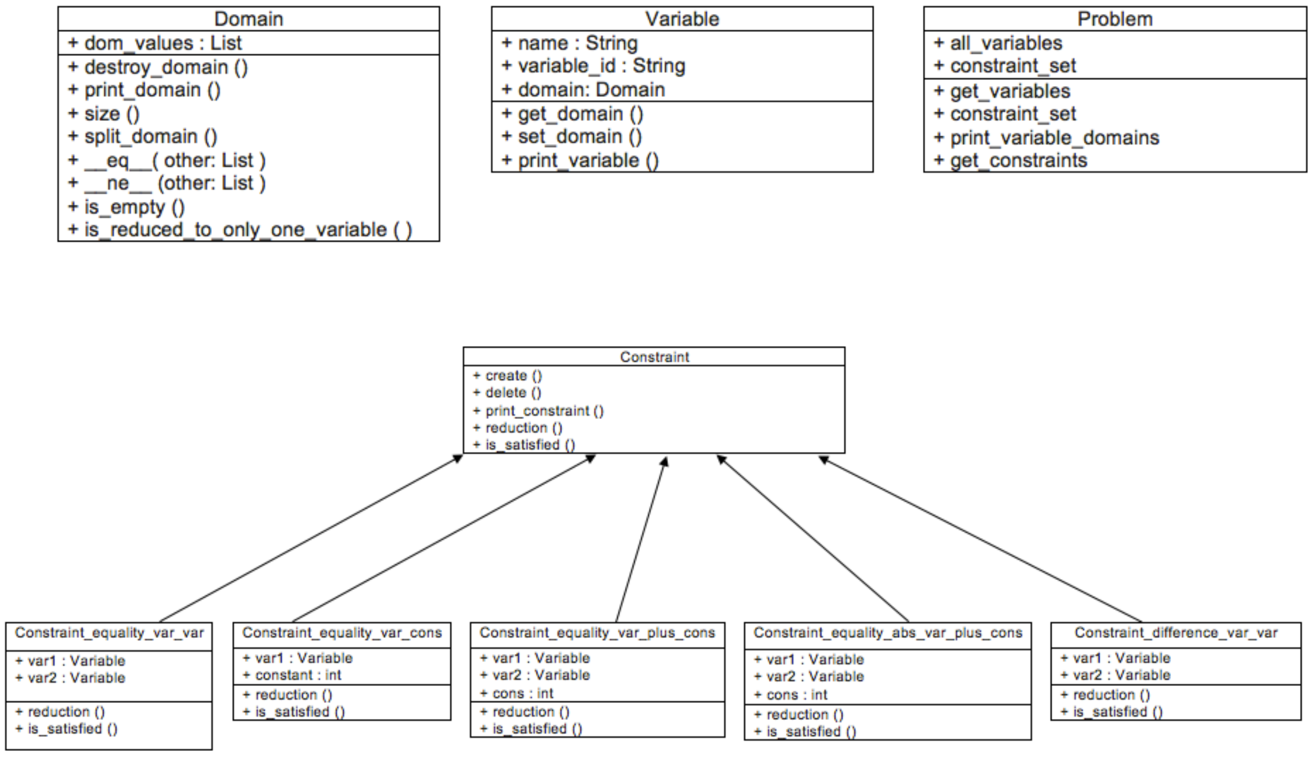

Figure 1: UML Diagrams

1.2 Classes and Data Structures

The problem is broken down into four main classes – Domain, Variable, Constraint and Problem. Each class aims to represent a part of the problem. Domain is the most base of the problem. Its purpose is to hold the possible values that a variable can take on (i.e. at the beginning all variables have possible values *'dom_values'* 1, 2, 3, 4, 5). Theses values are represented in a list data structure. Each domain object can be created, destroyed, printed, split in half and its size found by using the methods defined.

```
class Domain(object):
    def __init__(self, domain):
        self.dom_values = domain

    def destroy_domain(self):
        self.dom_values = None

    def print_domain(self):
        print(self.dom_values)

    def size(self):
        return len(self.dom_values)

    def split_domain(self):
        half = len(self.dom_values)//2
        return self.dom_values[:half], self.dom_values[half:]

    def __eq__(self, other):
        if self.dom_values == other.dom_values:
            return True
        else:
            return False

    def __ne__(self, other):
        if self.dom_values != other:
            return True
        else:
            return False

    def is_empty(self):
        return len(self.dom_values) == 0

    def is_reduced_to_only_one_variable(self):
        return len(self.dom_values) == 1
```

Figure 2: Structure of Domain Class

The Variable class holds all details corresponding to each variable in the problem. A Variable object has a name, id and domain – which takes the object Domain as described above. Each variable can be created, printed and its domain values can be retrieved and set.

```
class Variable(object):
    def __init__(self, name, variable_id, domain):
        self.create_variable(name, variable_id, domain)

    def create_variable(self, name, variable_id, domain):
        self.name = name
        self.variable_id = variable_id
        self.domain = domain

    def get_domain(self):
        return self.domain.dom_values

    def set_domain(self, dom):
        self.domain.dom_values = dom

    def print_variable(self):
        print("Variable Name: ", self.name)
        print("Variable id: ", self.variable_id)
        print("Domain: ", end=' ')
```

`        self.domain.print_domain()`
Figure 3: Structure of Variable Class

The Constraint class consists of a super class and five subclasses. The super class (called 'Constraint') defines the base structure for each of the following subclasses, Constraint has two abstract methods 'reduce' and 'is_satisfied'. The reduce method for each Constraint subclass reduces the possible values within the domain of the variable(s) under that constraint. The is_satisfied method ensures that the reduction in reduce has not violated the rules of the constraint. There are five different types of constraints *(see code project submission for implementation of each constraint)*:

- Constraint_equality_var_var - reduces the domains of two variables to their shared values
- Constraint_equality_var_cons - reduces a variables domain to the constant specified)
- Constraint_equality_var_plus_cons - reduces variable a and b's domains to values that satisfy the equation value a = value b + constant
- Constraint_equality_abs_var_plus_cons - reduces variable a and b's domains to values that satisfy that the absolute value of (value_a – value_b) equals a constant
- Constraint_difference_var_var – checks if either of the domains of two variables is reduced to one value and if so removes that value from the domain of the other variable

Each constraint generally iterates through the list of the possible values for a variable and either keeps, removes or changes these values mostly using list operations – e.g. append and remove.

As the problem has 25 variables, 25 variable objects are created. The appropriate constraint object is then created with the appropriate variable objects being passed as arguments to the constraint objects. There are 14 given constraints specified as well as the constraint that there must be only one type of each variable in each house (the types being nationality, pet, colour, beverage and board game).

The fourth class 'Problem' creates an object that holds all the variable objects (25) and constraint objects (64). This becomes useful when it comes to splitting domains later as it is necessary to make separate copies so as to try out different combinations of splitting the domains of a variable to finally get all variable domains to reduce down to 1 value each. The Problem class contains three methods – get_variables, set_variables and print_variable_domains.

2.0 Domain Reduction

This first step in implementing domain reduction is being able to iterate through the constraint list one by one and reduce the domains of that constraints variables. This step is performed by the domain_reduction function. This function iterates through the list of constraints calling the reduction method being performed on each constraint. Once this method has reduced the domain(s) of the variables accordingly a check is made on all constraints by calling the constraint method is_satisfied to make sure they are all still valid. This method checks that after the domain of a constraints variable(s) have been reduced, that all other constraints are still valid. If this check passes, the method moves onto the next constraint and the process is repeated. If the constraint is not satisfied (is_satisfied returns false) the loop is broken and the domain_reduction function returns false.

```
def domain_reduction(constraint_set):
    for const in constraint_set:
        const.reduction()
        for const2 in constraint_set:
            if const2.is_satisfied():
                print("Error not satisfied")
                return False
        else:
            return True
```

Figure 4: Function to iterate through each constraint once, perform the reduce method for each constraint and check that each constraint is still satisfied after each reduction

The next step was to perform domain reduction repeatedly until the domains could not be reduced any further. This is necessary because the reduction of the domains of one constraint can cause a chain effect of reductions within other constraints. The function 'reduce' performs this continuous loop of reductions for as long as necessary. The function consists of a 'while True' loop, within this loop the first step is to check the size of all variables domains using the domain_lengths function. The function from the first step, domain_reduction, is then called. If this function does not execute successfully the reduce terminates and returns a value of false. This implies that an error has occurred within the reduction of variable domains and a valid solution can no longer be found. After the domain_reduction function has completed, we check if a solution has been found – all domains contain 1 value – this is done b the function check_solution. If a solution is not found domain_lengths is called to once again check the size of the domains after reduction. If the size of the domains are the same before and after domain_reduction is performed, then the variable domains cannot be reduced any further and a solution must now be found through domain splitting.

```
def reduce(problem_set):
    constraints = problem_set.get_constraints()
    variables = problem_set.get_variables()
    while True:
        size1 = domain_lengths(variables)
        if not domain_reduction(constraints):
            return "error"
        size2 = domain_lengths(variables)
        #print("end: ", size2)
        if size1 == size2:
            return "split"
        if check_solution(variables):
            return "solution"
```

Figure 5: Function to reduce the all variables as far as possible

```
def check_solution(all_variables):
    for v in all_variables:
        if not v.domain.is_reduced_to_only_one_variable():
            return False
    else:
        return True
```

Figure 6: Function to check if a solution has been found

```
def domain_lengths(all_variables):
    domain_lengths = [None] * len(all_variables)
    i = 0
    for v in all_variables:
        domain_lengths[i] = v.domain.size()
        i += 1
    return domain_lengths
```

Figure 7: Function to check the lengths of the domains of all variables

3.0 Domain Splitting

The domain splitting function is set up to take a problem set and find a final solution to the problem by calling the function recursively on problem sets with updated domain values within the variables. The domain reduction function - reduce, described in section 2.0 - is then called. The result returned by the function reduce then determines the action taken by the split function. If the function reduce returns the string "*solution",* the check solution function within reduce has been satisfied (i.e. there is one single value in the domain of each variable). In this case the solution has been found and all variables are printed. If the reduce function returns the string *"error",* there is been a problem reducing the constraints and a solution cannot be found for that instance of the problem set. If the function reduce returns the string *"split"* the split function finds the size of all variables of the problem set passed as an argument to split. Then it finds the index of each variable that has a domain of length 2. For each of these variables, the domain is split in half (using the split_domain function from the Domain class). The split_domain function returns two lists, each containing half of the original domain. Two copies of the problem set are made and the variable whose domain has been split is updated to one of the lists returned (i.e. half of its original list of values) in each of the copies of the problem set. The function split is then called on each of the new copies of the problem set and the process is repeated. Therefore for each domain of length two, the function split s being recursively called twice.

```
def split(p_set):
    res = reduce(p_set)
    if res == "solution":
        p_set.print_variable_domains()
        return True
    elif res == "error":
        return False
    elif res == "split":
        set_vars = p_set.get_variables()
        size_domains = domain_lengths(set_vars)
        indices = [i for i,val in enumerate(size_domains) if val == 2]
        for ind in indices:
            x, y = set_vars[ind].domain.split_domain()
            set_1 = deepcopy(p_set)
            set_2 = deepcopy(p_set)
            v1 = set_1.get_variables()
            v1[ind].set_domain(x)
            v2 = set_2.get_variables()
            v2[ind].set_domain(y)
            return split(set_1), split(set_2)
```

Figure 8: Function to solve the problem – reduce domains as far as possible, split domains and recursively call the function again on each of the problem sets containing the split domains

4.0 Problems Encountered

When setting up domain splitting there was issues with changing types. When *dom_values* is originally set in the Domain class it is a list. Each variable within a constraint must have access to *dom_values*. The *dom_values* list is obtained via the *get_domain()* method in class Variable, this method returns the list of possible values remaining in a variables domain. When reduction is performed within a constraint the domain values for a variable are reduced and so the variables domain must be updated. To update the values the method *set_domain()* is used. Originally I had *set_domain()* setting the domain variable in the Variable object to be the list of remaining possible values of a variables domain. This was wrong because the variable domain within the class Variable should be of type Domain (referring to the class). This error meant that the first few iterations through the constraints would work correctly as get_domain was returning a list from the dom_values variable within Domain. But once a constraint came to a variable whose domain had already been updated the reduce method could not iterate through the variables domain as *get_domain()* was finding a List in the variable domain with the variable object. This caused an error as *get_domain()* could not be performed on a list object. This error was resolved by having *set_domain()* update the domain values of the variable *dom_values* within the Domain object for the variable.

ERROR	CORRECT
`set_domain()` → set variable domain in Variable to the list of remaining domain values	`set_domain()` → set variable *dom_values* in Domain from domain variable of class Variable

This caused great confusion as I first though it was an error in the implementation of the constraint's. After realizing the constraints were running for the first number of variables and that the error was occurring after a variables domain had been updated I was eventually able to find the solution.

Another issue was encountered when implementing domain splitting. Originally I was making a copy of the constraints and a copy of the variables separately and passing both of these as arguments to the split function. This was incorrect as the constraints and domains had to be copied together so that updates made to domains within the constraints would be reflected in the actual variables whose domains had been reduced, hence the whole problem set being copied now and passed into the split function.

Unfortunately, the function split does not split and reduce the domains down far enough to find a solution. The smallest size each of the domains reach are lengths:

[1, 1, 2, 1, 2, 3, 2, 3, 1, 1, 2, 1, 4, 2, 2, 2, 2, 1, 1, 3, 1, 1, 1, 1, 1]

The lengths of the last 5 variables domains are the colours, so their domains have all been reduced successfully. The final values remaining in each of the domains are as follows:

```
Variable Name:  english
Variable id:  nat1
Domain:  [3]
Variable Name:  spaniard
Variable id:  nat2
Domain:  [5]
Variable Name:  ukrainian
Variable id:  nat3
Domain:  [2, 4]
Variable Name:  norwegian
Variable id:  nat4
Domain:  [1]
Variable Name:  japanese
Variable id:  nat5
Domain:  [2, 4]
Variable Name:  zebra
Variable id:  pet1
Domain:  [1, 3, 4]
Variable Name:  snails
Variable id:  pet2
Domain:  [3, 4]
Variable Name:  fox
Variable id:  pet3
Domain:  [1, 3, 4]
Variable Name:  horse
Variable id:  pet4
Domain:  [2]
Variable Name:  dog
Variable id:  pet5
Domain:  [5]
Variable Name:  snakes and ladders
Variable id:  game1
Domain:  [3, 4]
Variable Name:  cluedo
Variable id:  game2
```

```
Domain:  [1]
Variable Name:  pictionary
Variable id:  game3
Domain:  [2, 3, 4, 5]
Variable Name:  travel the world
Variable id:  game4
Domain:  [2, 4]
Variable Name:  backgammon
Variable id:  game5
Domain:  [2, 4]
Variable Name:  orange juice
Variable id:  beverage1
Domain:  [2, 4]
Variable Name:  tea
Variable id:  beverage2
Domain:  [2, 4]
Variable Name:  milk
Variable id:  beverage3
Domain:  [3]
Variable Name:  coffee
Variable id:  beverage4
Domain:  [5]
Variable Name:  water
Variable id:  beverage5
Domain:  [1, 2, 4]
Variable Name:  red
Variable id:  colour1
Domain:  [3]
Variable Name:  green
Variable id:  colour2
Domain:  [5]
Variable Name:  ivory
Variable id:  colour3
Domain:  [4]
Variable Name:  blue
Variable id:  colour4
Domain:  [2]
Variable Name:  yellow
Variable id:  colour5
Domain:  [1]
```

Figure 9: results obtained after domain reduction and splitting

The split function does not find a solution as each resulting recursive call to split has returned False as the domains for that split functions problem set cannot be reduced any further. This could imply that there may be a slight error in how the domains are being split, possibly some small error in one of the constraints that is causing the domains to be reduced incorrectly.